DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings

Aayush Gupta Youngjae Kim Bhuvan Urgaonkar

Computer Systems Laboratory
Department of Computer Science and Engineering
The Pennsylvania State University, University Park, PA 16802
{axg354,youkim,bhuvan}@cse.psu.edu

Technical Report CSE-08-012
August 2008

Abstract

Recent technological advances in the development of flash-memory based devices have consolidated their leadership position as the preferred storage media in the embedded systems market and opened new vistas for deployment in enterprise-scale storage systems. Unlike hard disks, flash devices are free from any mechanical moving parts, have no seek or rotational delays and consume lower power. However, the internal idiosyncrasies of flash technology make its performance highly dependent on workload characteristics. The poor performance of random writes has been a cause of major concern which needs to be addressed to better utilize the potential of flash in enterprise-scale environments. We examine one of the important causes of this poor performance: the design of the Flash Translation Layer (FTL) which performs the virtual-to-physical address translations and hides the erase-before-write characteristics of flash. We propose a complete paradigm shift in the design of the core FTL engine from the existing techniques with our Demand-based Flash Translation Layer (DFTL) which selectively caches page-level address mappings. We develop and validate a flash simulation framework called FlashSim. Our experimental evaluation with realistic enterprise-scale workloads endorses the utility of DFTL in enterprise-scale storage systems by demonstrating: (i) improved performance, (ii) reduced garbage collection overhead and (iii) better overload behavior compared to state-of-the-art FTL schemes. For example, a predominantly random-write dominant I/O trace from an OLTP application running at a large financial institution shows a 78% improvement in average response time (due to a 3-fold reduction in operations of the garbage collector), compared to a state-of-the-art FTL scheme. Even for the well-known read-dominant TPC-H benchmark, for which DFTL introduces additional overheads, we improve system response time by 56%.

1 Introduction

Hard disk drives have been the preferred media for data storage in enterprise-scale storage systems for several decades. The disk storage market totals approximately $34 billion annually and is continually on the rise [41]. However, there are several shortcomings inherent to hard disks that are becoming harder to overcome as we move into faster and denser design regimes. Hard disks are significantly faster for sequential accesses than for random accesses and the gap continues to grow. This can severely limit the performance that hard disk based systems

are able to offer to workloads with significant random access component or lack of locality. In an enterprise-scale system, consolidation can result in the multiplexing of unrelated workloads imparting randomness to their aggregate [8].

Alongside improvements in disk technology, significant advances have also been made in various forms of solid-state memory such as NAND flash [1], magnetic RAM (MRAM) [33], phase-change memory (PRAM) [12], and Ferroelectric RAM (FRAM) [36]. Solid-state memory offers several advantages over hard disks: lower and more predictable access latencies for random requests, smaller form factors, lower power consumption, lack of noise, and higher robustness to vibrations and temperature. In particular, recent improvements in the design and performance of NAND flash memory (simply *flash* henceforth) have resulted in it being employed in many embedded and consumer devices. Small form-factor hard disks have already been replaced by flash memory in some consumer devices like music players, PDAs, digital cameras, etc. More recently flash drives with capacities of up to 256GB have also become available [1].

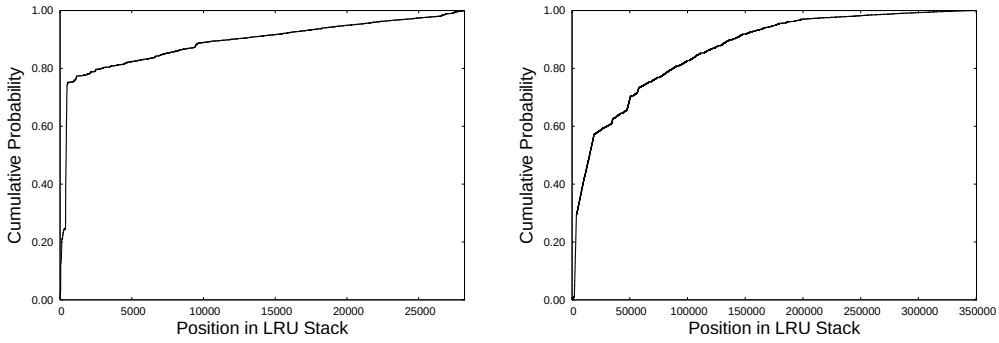
Flash devices are significantly cheaper than main memory technologies that play a crucial role in improving the performance of disk-based systems via caching and buffering. Furthermore, as an optimistic trend, their price-per-byte is falling [25], which leads us to believe that flash devices would be an integral component of future enterprise-scale storage systems. This trend is already evident as major storage vendors have started producing flash-based large-scale storage systems such as RamSan-500 from Texas Memory Systems, Symmetrix DMX-4 from EMC, etc. In fact, International Data Corporation has estimated that over 3 million Solid State Disks (SSD) will be shipped into enterprise applications, creating 1.2 billion dollars in revenue by 2011 [41].

Using Flash Memory in Enterprise-scale Storage. Before enterprise-scale systems can transition to employing flash-based devices at a large-scale, certain challenges must be addressed. It has been reported that manufacturers are seeing return rates of 20-30% on SSD-based notebooks due to failures and lower than expected performance [6]. While not directly indicative of flash performance in the enterprise, this is a cause for serious concern. Upon replacing hard disks with flash, certain managers of enterprise-scale applications are finding results that point to degraded performance. For example, recently Lee et al. [22] observed that "database servers would potentially suffer serious update performance degradation if they ran on a computing platform equipped with flash memory instead of hard disks." There are at least two important reasons behind this poor performance of flash for enterprise-scale workloads. First, unlike main memory devices (SRAMs and DRAMs), flash is *not always* superior in performance to a disk - in sequential accesses, disks might still outperform flash [22]. This

points to the need for employing hybrid storage devices that exploit the complementary performance properties of these two storage media. While part of our overall goal, this is out of the scope of this paper. The second reason, the focus of our current research, has to do with the *performance of flash-based devices for workloads with random writes.* Recent research has focused on improving random write performance of flash by adding DRAM-backed buffers [25] or buffering requests to increase their sequentiality [20]. However, we focus on an intrinsic component of the flash, namely the *Flash Translation Layer (FTL)* to provide a solution for this poor performance.

The Flash Translation Layer. The FTL is one of the core engines in flash-based SSDs that maintains a mapping table of virtual addresses from upper layers (e.g., those coming from file systems) to physical addresses on the flash. It helps to emulate the functionality of a normal block device by exposing only read/write operations to the upper software layers and by hiding the presence of *erase* operations, something unique to flash-based systems. Flash-based systems possess an asymmetry in how they can read and write. While a flash device can read any of its *pages* (a unit of read/write), it may only write to one that is in a special state called *erased*. Flashes are designed to allow erases at a much coarser spatial granularity than pages since page-level erases are extremely costly. As a typical example, a 16GB flash product from Micron [27] has 2KB pages while the erase blocks are 128KB . This results in an important idiosyncrasy of updates in flash. Clearly, in-place updates would require an erase-per-update, causing performance to degrade. To get around this, FTLs implement *out-of-place updates*. An out-of-place update: (i) chooses an already erased page, (ii) writes to it, (iii) invalidates the previous version of the page in question, and (iv) updates its mapping table to reflect this change. These out-of-place updates bring about the need for the FTL to employ a garbage collection (GC) mechanism. The role of the GC is to reclaim invalid pages within blocks by erasing the blocks (and if needed relocating any valid pages within them to new locations). Evidently, FTL crucially affects flash performance.

One of the main difficulties the FTL faces in ensuring high performance is the severely constrained size of the *on-flash SRAM-based cache* where it stores its mapping table. For example, a 16GB flash device requires at least 32MB SRAM to be able to map all its pages. With growing size of SSDs, this SRAM size is unlikely to scale proportionally due to the higher price/byte of SRAM. This prohibits FTLs from keeping virtual-to-physical address mappings for all pages on flash (page-level mapping). On the other hand, a block-level mapping, can lead to increased: (i) space wastage (due to internal fragmentation) and (ii) performance degradation (due to GC-induced overheads). To counter these difficulties, state-of-the-art FTLs take the middle approach of using a



(a) Financial Trace (b) TPC-C

Figure 1: The Cumulative Distribution Function (CDF) of virtual address access frequency obtained from (a) I/O trace from a financial institution [31] and (b) TPC-C benchmark [39] shows existence of significant temporal locality in I/O workloads. For the Financial trace, about 80% of the accesses belong to first 5000 requests in the LRU stack.

hybrid of page-level and block-level mappings and are primarily based on the following main idea (we explain the intricacies of individual FTLs in Section 2): most of the blocks (called Data Blocks) are mapped at the block level, while a small number of blocks called "update" blocks are mapped at the page level and are used for recording updates to pages in the data blocks.

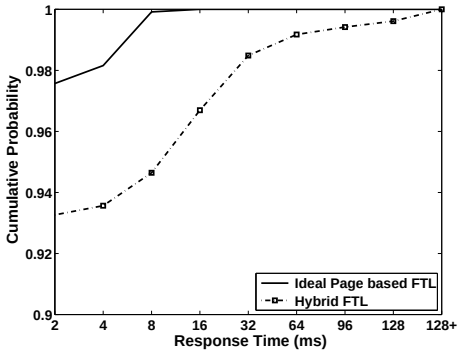


Figure 2: A comparison of the performance of a Financial trace employing an idealized page-level and a state-of-the-art hybrid FTL scheme.

As we will argue in this paper, various variants of hybrid FTL fail to offer good enough performance for enterprise-scale workloads. As a motivational illustration, Figure 2 compares the performance of a state-of-the-art hybrid FTL called FAST with an idealized page-level mapping scheme with sufficient flash-based SRAM. First, these hybrid schemes suffer from poor garbage collection behavior. Second, they often come with a number of workload-specific tunable parameters (for optimizing performance) that may be hard to set. Finally and most importantly, they do not properly exploit the temporal locality in accesses that most enterprise-scale workloads are known to exhibit. Figure 1 shows the extremely high temporal locality exhibited by two well-regarded workloads.

Even the small SRAM available on flash devices can thus effectively store the mappings in use at a given time while the rest could be stored on the flash device itself. Our thesis in this paper is that such a page-level FTL, based purely on exploiting such temporal locality, can outperform hybrid FTL schemes and also provide a easier-to-implement solution devoid of complicated tunable parameters.

Research Contributions. This paper makes the following specific contributions:

- We propose and design a novel Flash Translation Layer called *DFTL*. Unlike currently predominant hybrid FTLs, it is purely page-mapped. The idea behind DFTL is simple: since most enterprise-scale workloads exhibit significant temporal locality, DFTL uses the on-flash limited SRAM to store the most popular (specifically, most recently used) mappings while the rest are maintained on the flash device itself. The core idea of DFTL is easily seen as inspired by the intuition behind the Translation Lookaside Buffer (TLB) [10].
- We implement an accurate flash simulator called *FlashSim* to evaluate the efficacy of DFTL and compare it with other FTL schemes. FlashSim is open-source and is built by enhancing the popular Disksim [7] simulator. Flashsim simulates the flash memory, controller, caches, device drivers and various interconnects.
- Using a number of realistic enterprise-scale workloads, we demonstrate the improved performance resulting from DFTL. As illustrative examples, we observe 78% improvement in average response time for a random write-dominant I/O trace from an OLTP application running at a large financial institution and 56% improvement for the read-dominant TPC-H workload.

The rest of this paper is organized as follows. In Section 2, we present the basics of flash memory technology including a classification of various existing FTL schemes. The design of DFTL and its comparison with hybrid FTL schemes is described in Section 3. Section 4 describes the framework of our simulator FlashSim. Experimental results are presented in Section 5. The conclusions of this study are described in Section 6.

2 Background and Related Work

Basics of Flash Memory Technology. Recently, significant advances have been made in various forms of solid-state memory such as NAND flash [34], magnetic RAM (MRAM) [33], phase-change memory (PRAM), and FRAM [**?**]. In particular, improvements in the design and performance of NAND flash memory (simply flash henceforth) have resulted in it being employed in many embedded and consumer devices. Small form-factor hard disks have already been replaced by flash memory in some consumer devices, like music players. More recently,

flash drives with capacities in the 32-64 GB range have become available and have been used in certain laptops as the secondary storage media [35].

Flash is a unique storage device since unlike the hard disk drive and volatile memories, which provide read and write opertations, it also provides an *erase operation* [29]. flash provides three basic operations: (i) program or write, (ii) read, and (iii) erase. Salient operational characteristics of these operations are as follows [29]. The write operation changes the value of a bit in a flash memory cell from 1 to 0. The erase operation changes a bit from 0 to 1. Single bit erase operations are not typically supported. Erase operations are performed at the granularity of a *block* (a set of contiguous bits). changing all the bits of the block to 1. Erase is the slowest operation while write is slower than read. The life-time of flash memory is limited by the number of erase operations. It has been reported that each flash memory cell can sustain about 10K-100K erase operations [29]. Moreover, flash memory can be composed of two types of memory cells: Single-Level-Cell (SLC) which stores one bit per cell and Multi-Level-Cell (MLC), introduced by Intel [13], which stores multiple bits of data per memory cell [13]. However, improving the density of flash memory using MLC has been found to deteriorate its lifetime and performance [13]. In our research, we focus on SLC based flash memory. This tension between cost, reliability, and performance is likely to continue in the foreseeable future.

An erase unit, is composed of multiple *pages*. A page is the granularity at which reads and writes are performed. In addition to its data area, a page contains a small spare Out-of-Band area (OOB) which is used for storing a variety of information including: (i) Error Correction Code (ECC) information used to check data correctness, (ii) the logical page number corresponding to the data stored in the data area and (iii) page state. Each page on flash can be in one of three different states: (i) *valid*, (ii) *invalid* and (iii) *free/erased*. When no data has been written to a page, it is in the erased state. A write can be done only to an erased page, changing its state to valid. When data is written to an erased page, its state becomes valid. If the page contains an older version of data, it is said to be in the invalid state. As was pointed out, out-of-place updates result in certain written pages whose entries are no longer valid. They are called invalid pages. Flash comes as a *small block* or *large block* device. Using fewer blocks not only improves read, write, and erase performance, but also reduces chip size by reducing gaps between blocks [38]. A small block scheme can have 8KB or 16KB blocks where each page contains 512B data area and 16B OOB. On the contrary, large block schemes have 32KB to 128KB blocks where each page contains 2KB data area and 64B OOB. Table 1 shows detailed organization and performance characteristics for these two variants of state-of-the-art flash devices [38].

Flash Type	Data Unit Size			Access Time		
	Page (Bytes)		Block (Bytes)	Page READ (us)	Page WRITE (us)	Block ERASE (ms)
	Data Area	OOB Area				
Small Block	512	16	(16K+512)	41.75	226.75	2
Large Block	2048	64	(128K+4K)	130.9	405.9	2

Table 1: NAND Flash organization and access time comparison for Small-Block vs. Large-Block schemes [38].

2.1 Characteristics of Flash Memory Operations

In this subsection, we describe key operational characteristics of flash memory.

- **Asymmetric operation speeds -** Flash memory includes the following operational characteristics: Basically the read and write speed of flash memory is asymmetric. Not only are erase operations done at the coarser granularity of a block and are significantly slower that reads/writes, there is an additional asymmetry between access times of reads and writes. As shown in Table 1, erase operations are significantly slower than reads/writes. Additionally, write latency can be higher than read latency by up to a factor of 4-5. This is because draining electrons from a flash cell for a write takes longer than sensing them for a read. Note that this is significantly different from hard disk and volatile memory.
- **Out-of-place updates -** In flash memory, in-place update operations are very costly. Since an erase occurs at the block granularity whereas writes are done to pages, an in-place update to a page entails (i) reading all valid pages of the block into a buffer, (ii) updating the required page, (iii) erasing the entire block and (iv) then writing back all the valid pages to the block. Instead, faster out-of-place updates are employed that work as follows: An out-of-place update invalidates the current version of the page being updated and writes the new version to a free page. This introduces the need to keep track of the current page version location on flash itself, which is maintained by implementing an address translation layer (FTL). the FTL. The OOB area of invalid pages are marked to indicate their changed states.
- **Garbage collection -** Out-of-place updates result in the creation of invalid pages on flash. A garbage collector is employed to reclaim invalid pages and create new erased blocks. It first selects a victim block based on a policy such as choosing a block with maximum invalid pages. All valid data within the block is first copied into an erased block. This data-rewrite operation can be quickly and efficiently processed by the special support of a *Copy-Back Program* operation where an entire page is moved into the internal data buffer first Then the victim block is erased. The efficiency of garbage collector is one of the dominant factors affecting flash memory performance.

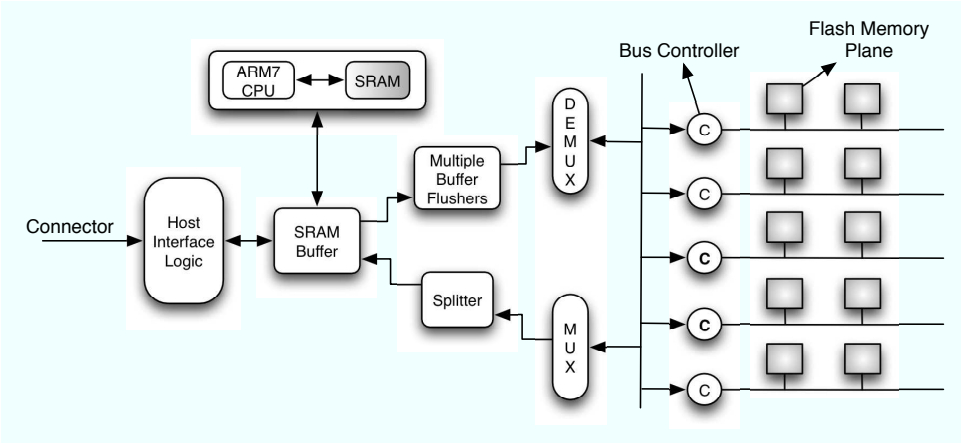


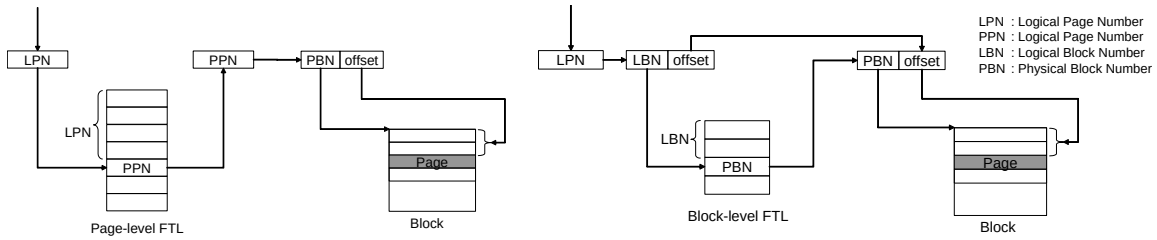
Figure 3: A Block Diagram of Flash based Solid State Disk Drive.

- **Wear-leveling -** The lifetime of flash memory is limited by the number of erase operations on its cells. Each memory cell typically has a lifetime of 10K-1M erase operations [5]. Thus, *wear-leveling* techniques [15, 19, 26] are used to delay the wear-out of the first flash block. As we will see, our new mapping scheme will improve flash performance by improved GC. to make flash memory last longer by evenly distributing the wear-out over all blocks. Since the MLC has much smaller voltage tolerance than SLC, each write-erase cycle tends to increase the variance in the voltage stored. Thus, the lifetime of MLC is more limited than that of SLC [13]. whereas data density is more in MLC and benefited in cost by aroud two times more than SLC [13]. Also, it degrades the performance will need to differentiate data more precisely.

  The granularity at which wear-leveling is carried out impacts the variance in the lifetime of individual blocks and also the performance of flash. The finer the granularity, the smaller the variance in lifetime. However, it may impose certain performance overheads. Thus, this trade-off needs to be balanced to obtain optimal performance from the device while providing the minimal desired lifetime. The optimal selection for the victim block considers the copy overhead of valid pages and the wear-level of the block for wear-leveling.

2.2 Flash Memory based Solid State Disk Drive

A solid state disk-drive (SSD) can be composed of non-volatile memory such as battery-backed DRAM (DRAM-Based SSDs) or flash memory chips (Flash-based SSDs). There are also hybrid devices incorporating DRAM and flash memory [37]. RamSan-500, a cached flash SSD from Texas Memory Systems (TMS) is a hybrid of DDR RAM and NAND-SLC Flash Memory [32]. Symmetrix DMX-4 from EMC [37] is an enterprise networked storage system employing the flash drives. Since NAND flash memory based SSD has recently become popular,



(a) Page-level FTL Scheme (b) Block-level FTL Scheme

Figure 4: (a)-(b) Page-level and Block-level FTL schemes. LPN: Logical Page Number, PPN: Physical Page Number, LBN: Logical Block Number, PBN: Physical Block Number.

in this work we only concentrate on NAND flash memory based SSD.

Figure 3 describes the organization of internal components in a Flash-Based SSD [28]. It possesses a host interface such as Fiber-Channel, SATA, PATA, and SCSI etc. to appear as block I/O device to the host computer. The main controller is composed of two units - processing unit (such as ARM7 processor) and fast access memory (such as SRAM). The virtual-to-physical mappings are processed by the processor and the data-structures related to the mapping table are stored in SRAM in the main controller. The software module related to this mapping process is called Flash Translation Layer (FTL). A part of SRAM is also used for caching data.

A storage pool in a SSD is composed of multiple flash memory *Planes*. The *Planes* are implemented in multiple *Dies*. For example, Samsung 4GB flash memory has two *Dies*. A *Die* is composed of four planes, each of size is 512MB [30]. A *Plane* consists of a set of blocks. The block size can be 64KB, 128KB, 256KB etc. depending on the memory manufacturer. The SSD can be implemented multiple *Planes*. SSD performance can be enhanced by interleaving requests across the planes [30], which is achieved by the multiplexer and de-multiplexer between SRAM buffer and flash memories. In our research, we deal with a simplistic, SLC flash device model with block size as 128KB.

Details of Flash Translation Layer. The mapping tables and other data structures, manipulated by the FTL are stored in a small, fast SRAM. The FTL algorithms are executed on it. FTL helps in emulating flash as a normal block device by performing out-of-place updates which in turn helps to hide the erase operations in flash. It can be implemented at different address translation granularities. At two extremes are page-level and block-level translation schemes which we discuss next. As has been stated, We begin by understanding two extremes of FTL designs with regard to what they store in their in-SRAM mapping table. Although neither is used in practice, these will help us understand the implications of various FTL design choices on performance.

Page-level and Block-level FTL Schemes. As shown in Figure 4(a), In a page-level FTL scheme, the logical page number of the request sent to the device from the upper layers such as file system can be mapped into any page within the flash. This should remind the reader of a fully associative cache [10]. Thus, it provides compact and efficient utilization of blocks within the flash device. However, on the downside, such translation requires a large mapping table to be stored in SRAM. For example, a 16GB flash memory requires approximately 32MB of SRAM space for storing a page-level mapping table. Given the order of magnitude difference in the price/byte of SRAM and flash; having large SRAMs which scale with increasing flash size is infeasible.

At the other extreme, in a block-level FTL scheme, as depicted in Figure 4(b), page offset within a block is fixed. the logical block number is translated into a physical block number using the mapping table similar to set-associative cache design [10]. The logical page number offset within the block is fixed. Figure 4-(b) shows an example of block-based address translation. The Logical Page Number (LPN) is converted into a Logical Block Number(LBN) and offset. The LBN is then converted to Physical Block Number (PBN) using the block based mapping table. Thus, the offset within the block is invariant to address translation. The size of the mapping table is reduced by a factor of *block size/page size* (128KB/2KB=64) as compared to page-level FTL. However, it provides less flexibility as compared to the page-based scheme. However, even if there are free pages within a block except at the required offset, this scheme may require allocation of another free block; thus reducing the efficiency of block utilization. However, since a given logical page may now be placed in only a particular physical page within each block, the possibility of finding such a page decreases. As a result the garbage collection overheads grow. Moreover, the specification for large block based flash devices requiring sequential programming within the block [38] making this scheme infeasible to implement in such devices. For example, Replacement Block-scheme [3] is a block-based FTL scheme in which each data block is allocated replacement blocks to store the updates. The chain of replacement blocks along with the original data block are later merged during garbage collection. Figure 4 illustrates the differences in address translation between these two schemes.

A Generic Description of Hybrid FTL Scheme. To address the shortcomings of the above two extreme mapping schemes, researchers have come up with a variety of alternatives. Log-buffer based FTL scheme is a hybrid FTL which combines a block-based FTL with a page-based FTL as shown in Figure 5.

The entire flash memory is partitioned into two types of blocks - *Data* and *Log/Update* blocks. First write to a logical address is done in data blocks. Although many schemes have been proposed [14, 4, 23, 16, 24], they share one fundamental design principle. All of these schemes are a *hybrid* between page-level and block-level schemes.

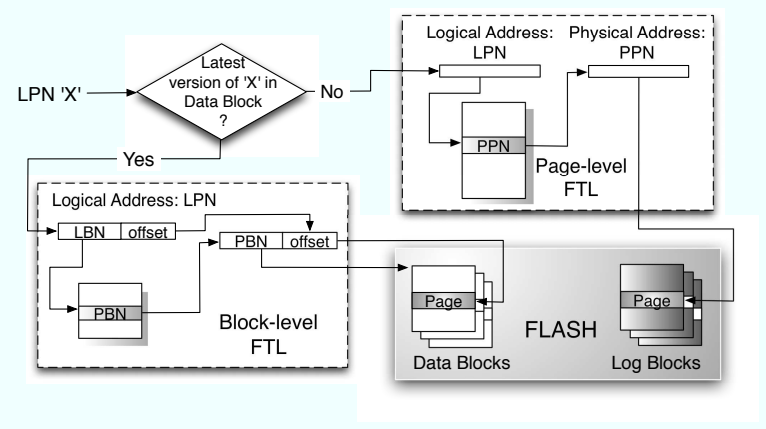


Figure 5: Hybrid FTL Scheme, combining a block-based FTL for data blocks with a page-based FTL for log blocks. LPN: Logical Page Number, PPN: Physical Page Number, LBN: Logical Block Number, PBN: Physical Block Number.

They logically partition their blocks into two groups - *Data Blocks* and *Log/Update Blocks*. Data blocks form the majority and are mapped using the block-level mapping scheme. A second special type of blocks are called log blocks whose pages are mapped using a page-level mapping style. Figure **??** illustrates such hybrid FTLs. Any update on the data blocks are performed by writes to the log blocks. The log-buffer region is generally kept small in size (for example, 3% of total flash size [24]) to accommodate the page-based mappings in the small SRAM. Extensive research has been done in optimizing log-buffer based FTL schemes [14, 4, 23, 16, 24].

Garbage Collection in Hybrid FTLs. The hybrid FTLs invoke a garbage collector whenever no free *log blocks* are available. Garbage Collection requires merging log blocks with data blocks. The merge operations can be classified into: *Switch merge*, *Partial merge*, and *Full merge*.

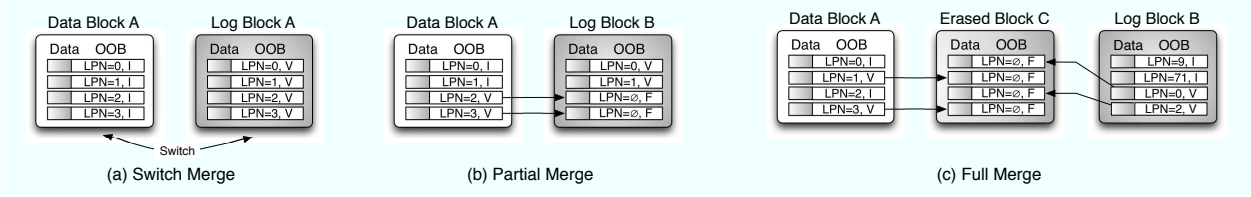


Figure 6: Various *Merge* operations (*Switch*, *Partial*, and *Full*) in log-buffer based FTL schemes. V: Valid, I: Invalid, and F: Free/Erased and LPN is Logical Page Number.

In Figure 6(a), since log block B contains all valid, sequentially written pages corresponding to data block A, a simple *Switch Merge* is performed, whereby log block B becomes new data block and the old data block A is erased. Figure 6(b) illustrates *Partial Merge* between block A and B where only the valid pages in data block A

are copied to log block B and the original data block A is erased changing the block B's status to a data block. *Full Merge* involves the largest overhead among the three types of merges. As shown in Figure 6(c), Log block B is selected as the victim block by the garbage collector. The valid pages from the log block B and its corresponding data block A are then copied into a new erased block C and block A and B are erased. However, full merge can become a long recursive operation in case of a fully-associative log block scheme where the victim log block has pages corresponding to multiple data blocks and each of these data blocks have updated pages in multiple log blocks.

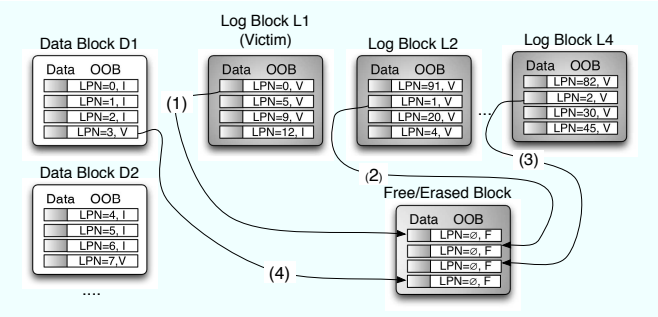


Figure 7: Expensive Full Merge.

This situation is illustrated in Figure 7. log block L1 containing randomly written data is selected as a victim block for garbage collection. It contains valid pages belonging to data blocks D1, D2 and D3. An erased block is selected from the free block pool and the valid pages belonging to D1 are copied to it from different log blocks and D1 itself in the order shown. The data block D1 is then erased. Similar operations are carried out for data blocks D2 & D3 since L1 contains the latest version of some of the pages for these blocks. Finally, log block L1 is erased. *Thus, random writes in hybrid FTLs induce costly garbage collection which in turn affects performance of subsequent operations irrespective of whether they are sequential or random.* Recent log buffer-based FTL schemes [16, 24] have tried to reduce the number of these full merge operations by segregating log blocks based on access patterns. Hot blocks with frequently accessed data generally contain large number of invalid pages whereas cold blocks have least accessed data. Utilizing hot blocks for garbage collection reduces the valid page copying overhead, thus lowering the full merge cost.

State-of-the-art FTLs. State-of-the-art FTLs [4, 23, 16, 24] are based on hybrid log-buffer based approaches. They try to address the problems of expensive full merges, which are inherent to any log-buffer based hybrid scheme, in their own unique way. However, all of these attempts are unable to provide the desired results.

- Block Associative Sector Translation (BAST) [4] scheme exclusively associates a log block with a data block. In presence of small random writes, this scheme suffers from *log block thrashing* [23] that results in increased full merge cost due to inefficiently utilized log blocks.
- Fully Associative Sector Translation (FAST) [23] allows log blocks to be shared by all data blocks. This improves the utilization of log blocks as compared to BAST. FAST keeps a single sequential log block dedicated for sequential updates while other log blocks are used for performing random writes. Thus, it cannot accommodate multiple sequential streams. Further, it does not provide any special mechanism to handle temporal locality in random streams.
- SuperBlock FTL [16] scheme utilizes existence of *block level* spatial locality in workloads by combining consecutive logical blocks into a superblock. It maintains page-level mappings within the superblock to exploit temporal locality in the request streams by separating hot and cold data within the superblock. However, the three-level address translation mechanism employed by this scheme causes multiple OOB area reads and writes for servicing the requests. More importantly, it utilizes a fixed superblock size which needs to be explicitly tuned to adapt to changing workload requirements.
- The recent Locality-Aware Sector Translation (LAST) scheme [24] tries to alleviate the shortcomings of FAST by providing multiple sequential log blocks to exploit spatial locality in workloads. It further separates random log blocks into hot and cold regions to reduce full merge cost. In order to provide this dynamic separation, LAST depends on an external locality detection mechanism. However, Lee et al. [24] themselves realize that the proposed locality detector cannot efficiently identify sequential writes when the small-sized write has a sequential locality. Moreover, maintaining sequential log blocks using a block-based mapping table requires the sequential streams to be aligned with the starting page offset of the log block in order to perform switch-merge. Dynamically changing request streams may impose severe restrictions on the utility of this scheme to efficiently adapt to the workload patterns.

3 Design of DFTL: Our Demand-based Page-mapped FTL

We have seen that any hybrid scheme, however well-designed or tuned, will suffer performance degradation due to expensive full merges that are caused by the difference in mapping granularity of data and update blocks. *Our contention is that a high-performance FTL should completely be re-designed by doing away with log-blocks.* Demand-based Page-mapped FTL (DFTL) is an enhanced form of the page-level FTL scheme described in Sec-

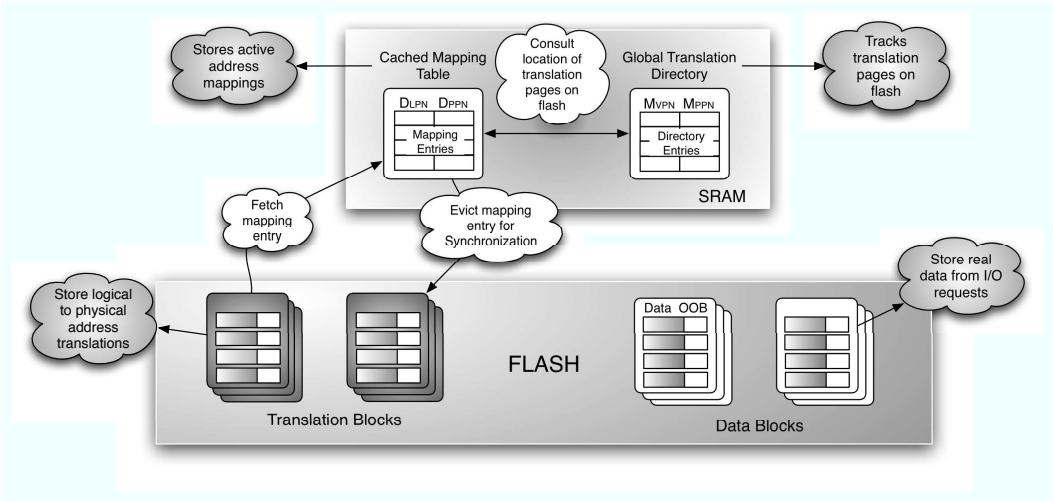


Figure 8: Schematic Design of DFTL. D_{LPN}: Logical Data Page Number, D_{PPN}: Physical Data Page Number, M_{VPN}: Virtual Translation Page Number, M_{PPN}: Physical Translation Page Number.

tion 2. It does away completely with the notion of log blocks. In fact, all blocks in this scheme, can be used for servicing update requests. Page-level mappings allow requests to be serviced from any physical page on flash. However, to make the fine-grained mapping scheme feasible with the constrained SRAM size, a *special address translation mechanism* has to be developed. In the next sub-sections, we describe the architecture and functioning of DFTL and highlight its advantages over existing state-of-the-art FTL schemes.

3.1 DFTL Architecture

DFTL makes use of the presence of temporal locality in workloads to judiciously utilize the small on-flash SRAM. Instead of the traditional approach of storing all the address translation entries in the SRAM, it dynamically loads and unloads the page-level mappings depending on the workload access patterns. Furthermore, it maintains the complete image of the page-based mapping table on the flash device itself. There are two options for storing the image: (i) The OOB area or (ii) the data area of the physical pages. We choose to store the mappings in the data area instead of OOB area because it enables us to group a larger number of mappings into a single page as compared to storing in the OOB area. For example, if 4 Bytes are needed to represent the physical page address in flash, then we can group 512 logically consecutive mappings in the data area of a single page whereas only 16 such mappings would fit an OOB area. Moreover, the additional space overhead incurred is negligible as compared to the total flash size. A 1GB flash device requires only about 2MB (approximately 0.2% of 1GB) space for storing all the mappings.

Data Pages and Translation Pages. In order to store the address translation mappings on flash data area, we segregated *Data-Pages* and *Translation-Pages*. Data pages contain the real data which is accessed or updated during read/write operations whereas pages which only store information about logical-to-physical address mappings are called as translation pages. Blocks containing translation pages are referred to as *Translation-Blocks* and *Data-Blocks* store only data pages. As is clear from Figure 8, translation blocks are totally different from log blocks and are only used to store the address mappings. They require only about 0.2% of the entire flash space and do not require any merges with data blocks.

3.2 Logical to Physical Address Translation

A request is serviced by reading from or writing to pages in the data blocks while the corresponding mapping updates are performed in translation blocks. We describe various data structures and mechanisms required for performing address translation and discuss their impact on the overall performance of DFTL.

Global Mapping Table and Global Translation Directory. The entire logical-to-physical address translation set is always maintained on some logically fixed portion of flash and is referred to as the *Global Mapping Table*. However, only a small number of these mappings can be present in SRAM. These active mappings present in SRAM form the *Cached Mapping Table (CMT)*. Since out-of-place updates are performed on flash, translation pages get physically scattered over the entire flash memory. DFTL keeps track of all these translation pages on flash by using a *Global Translation Directory (GTD)*. Although GTD is permanently maintained in the SRAM, it does not pose any significant space overhead. For example, for a 1GB flash memory device, 1024 translation pages are needed (each capable of storing 512 mappings), requiring a GTD of about 4KB.

DFTL Address Translation Process. Algorithm 1 describes the process of address translation for servicing a request. If the required mapping information for the given read/write request exists in SRAM (in CMT), it is serviced directly by reading/writing the data page on flash using this mapping information. If the information is not present in SRAM then it needs to be fetched into the CMT from flash. However, depending on the state of CMT and the replacement algorithm being used, it may entail evicting entries from SRAM. We use the segmented LRU array cache algorithm [18] for replacement in our implementation. However, other algorithms such as evicting Least Frequently Used mappings can also be used.

If the victim chosen by the replacement algorithm has not been updated since the time it was loaded into

SRAM, then the mapping is simply erased without requiring any extra operations. This reduces traffic to translation pages by a significant amount in read-dominant workloads. In our experiments, approximately 97% of the evictions in read-dominant TPC-H benchmark did not incur any eviction overheads. Otherwise, the Global Translation Directory is consulted to locate the victim's corresponding translation page on flash. The page is then read, updated, and re-written to a new physical location. The corresponding GTD entry is updated to reflect the change. Now the incoming request's translation entry is located using the same procedure, read into the CMT and the requested operation is performed. The example in Figure 9 illustrates the process of address translation when a request incurs a CMT miss.

Input: Request's Logical Page Number ($request_{lpn}$), Request's Size ($request_{size}$)
Output: NULL
while $request_{size} \neq 0$ **do**
  if $request_{lpn}$ *miss in Cached Mapping Table* **then**
    if *Cached Mapping Table is full* **then**
      /* Select entry for eviction using segmented LRU replacement algorithm */
      $victim_{lpn} \leftarrow \mathbf{select\_victim\_entry}()$
      if $victim_{last\_mod\_time} \neq victim_{load\_time}$ **then**
        /*$victim_{type}$: Translation or Data Block
        $Translation\_Page_{victim}$: Physical Translation-Page Number containing victim entry */
        $Translation\_Page_{victim} \leftarrow \mathbf{consult\_GTD}\ (victim_{lpn})$
        $victim_{type} \leftarrow$ Translation Block
        $\mathbf{DFTL\_Service\_Request}(victim)$
      end
      $\mathbf{erase\_entry}(victim_{lpn})$
    end
    $Translation\_Page_{request} \leftarrow \mathbf{consult\_GTD}(request_{lpn})$
    /* Load map entry of the request from flash into Cached Mapping Table */
    $\mathbf{load\_entry}(Translation\_Page_{request})$
  end
  $request_{type} \leftarrow$ Data Block
  $request_{ppn} \leftarrow \mathbf{CMT\_lookup}(request_{lpn})$
  $\mathbf{DFTL\_Service\_Request}(request)$
  $request_{size}$- -
end

Algorithm 1: DFTL Address Translation

Overhead in DFTL Address Translation. The worst-case overhead includes two translation page reads (one for the victim chosen by the replacement algorithm and the other for the original request) and one translation page write (for the victim) when a CMT miss occurs. However, our design choice is rooted deeply in the existence of temporal locality in workloads which helps in reducing the number of evictions. Furthermore, the presence of multiple mappings in a single translation page allows *batch updates* for the entries in the CMT, physically co-located with the victim entry. We later show through detailed experiments that the extra overhead involved with address translation is much less as compared to the benefits accrued by using a fine-grained FTL.

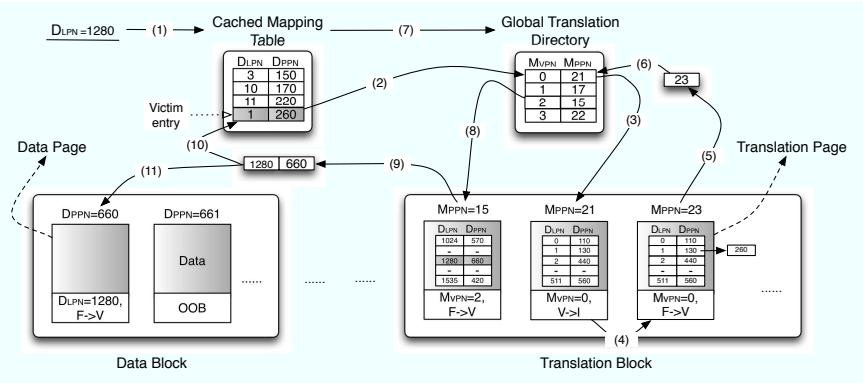


Figure 9: (1) Request to D_{LPN} 1280 incurs a miss in Cached Mapping Table (CMT), (2) Victim entry D_{LPN} 1 is selected, its corresponding translation page M_{PPN} 21 is located using Global Translation Directory (GTD), (3)-(4) M_{PPN} 21 is read, updated (D_{PPN} 130 $\rightarrow$ D_{PPN} 260) and written to a free translation page (M_{PPN} 23), (5)-(6) GTD is updated (M_{PPN} 21 $\rightarrow$ M_{PPN} 23) and D_{LPN} 1 entry is erased from CMT. (7)-(11) The original request's (D_{LPN} 1280) translation page is located on flash (M_{PPN} 15). The mapping entry is loaded into CMT and the request is serviced. Note that each GTD entry maps 512 logically consecutive mappings.

3.3 Read/Write Operation and Garbage Collection

Read requests are directly serviced through flash page read operations once the address translation is completed. DFTL maintains two blocks, namely *Current Data Block* and *Current Translation Block*, where the data pages and translation pages are written, respectively. Page-based mappings allow sequential writes within these blocks, thus conforming to the large-block sequential write specification [38]. For write requests, DFTL allocates the next available free page in the Current Data Block, writes to it and then updates the map entry in the CMT.

However, as writes/updates propagate through the flash, over a period of time the available physical blocks (in erased state) decreases. DFTL maintains a high watermark called $GC_{threshold}$, which represents the limit till which writes are allowed to be performed without incurring any overhead of garbage collection for recycling the invalidated pages. Once $GC_{threshold}$ is crossed, DFTL invokes the garbage collector. Victim blocks are selected based on a simple cost-benefit analysis that we adapt from [19].

Different steps are followed depending on whether the victim is a translation block or a data block before returning it to the free block pool after erasing it. If it is a translation block, then we copy the valid pages to the Current Translation Block and update the GTD. However, if the victim is a data block, we copy the valid pages to the Current Data Block and update all the translation pages and CMT entries associated with these pages. In order to reduce the operational overhead, we utilize a combination of *lazy copying* and *batch updates*. Instead of updating the translation pages on flash, we only update the CMT for those data pages whose mappings are present in it. This technique of *lazy copying* helps in delaying the proliferation of updates to flash till the corresponding

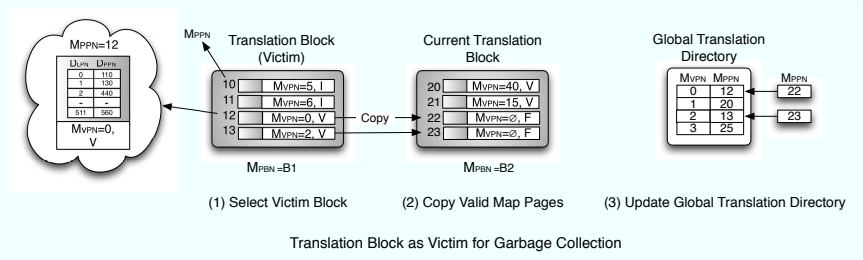


Figure 10: **Example:** (1) Translation Block (M_{PBN} B1) is selected as Victim for Garbage Collection. (2) Valid pages M_{PPN} 12 & M_{PPN} 13 are copied to the *Current Translation Block* (M_{PBN} B2) at free pages M_{PPN} 22 & M_{PPN} 23. (3) Global Translation Directory entries corresponding to M_{VPN} 0 & M_{VPN} 2 are updated (M_{PPN} 12 $\rightarrow$ M_{PPN} 22, M_{PPN} 13 $\rightarrow$ M_{PPN} 23.

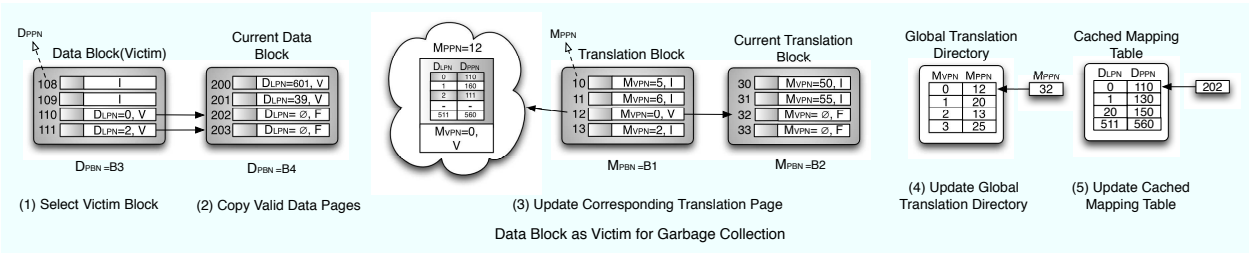


Figure 11: **Example:** (1) Data Block (D_{PBN} B3) is selected as Victim for Garbage Collection. (2) Valid pages D_{PPN} 110 & D_{PPN} 111 are copied to the *Current Data Block* (D_{PBN} B4) at free pages D_{PPN} 202 & D_{PPN} 203. (3) Translation page M_{PPN} 12 containing the mappings for the valid pages D_{PPN} 110 & D_{PPN} 111 is updated and copied to the Current Map Block (M_{PBN} B2). (4) Global Translation Directory entry corresponding to M_{VPN} 0 is updated (M_{PPN} 12 $\rightarrow$ M_{PPN} 32). (5) Since D_{LPN} 0 is present in Cached Mapping Table, the entry is also updated (D_{PPN} 110 $\rightarrow$ D_{PPN} 202). Note: We do not illustrate the advantages of batch updates and lazy copying in this example.

mappings are evicted from SRAM. Moreover, multiple valid data pages in the victim may have their virtual-to-physical address translations present in the same translation-page. By combining all these modifications into a single *batch update*, we reduce a number of redundant updates. The associated Global Translation Directory entries are also updated to reflect the changes. The example in Figure 11 displays the working of our Garbage Collector when the $GC_{threshold}$ is reached and a data block is selected as victim. Owing to space constraints, we do not present algorithms for garbage collection and overall read/write operations.

```
Input: NULL
Output: NULL
victim ← select_victim_entry();
/* Victim is TRANSLATION BLOCK */
if victim_type ∈ TRANSLATION_BLOCK_SET then
    foreach victim_page(i) do
        /* (i) Copy only valid pages in the victim block to the Current Translation Block, (ii) invalidate
        old pages, and update Global Translation Directory */
        if victim_page(i) is valid then
            curr_translation_blk ← get_curr_translation_blk();
            copy_page(victim_page(i), curr_map_blk);
            update_GTD(victim_page(i));
        end
    end
end
else
    /* Victim is DATA BLOCK */ foreach victim_page(i) do
        /* Copy only valid pages in the victim block to the
        current data block, invalidate old pages, and mark
        their corresponding translation pages for update */
        if victim_page(i) is valid then
            curr_data_blk ← get_curr_data_blk();
            copy_page(victim_page(i), curr_data_blk);
            translation_page_update_set[] ← mark_corr_translation_page_for_update (victim_page(i));
        end
    end
    /* perform batch update on the marked Translation Pages */
    foreach translation_page_i ∈ translation_page_update_set do
        curr_translation_blk ← get_curr_translation_blk();
        old_translation_page ← translation_page_i;
        update_translation_page(translation_page_i, curr_translation_blk);
        invalidate(old_translation_page);
        update_GTD(translation_page_i);
        if translation_page_i ∈ CachedMappingTable then
            update_CMT(translation_page_i);
        end
    end
end
erase_blk(victim); /* erase the victim block */
```

Algorithm 2: Garbage Collection

3.4 Dealing with Power Failure

Although flash is a non-volatile storage device, it relies on volatile on-flash SRAM which is susceptible to power failure in the host. When power failure occurs, all logical-physical mapping information stored in the Cached Mapping Table on SRAM will be lost. The traditional approach of reconstructing the mapping table utilizes scanning the logical addresses stored in the OOB area of all physical pages on flash [23]. However, the scanning process incurs high overhead and leads to long latencies while the mapping table is being recovered. In DFTL, the Global Translation Directory stores the locational information corresponding to the Global Mapping Table. Thus, storing the GTD on non-volatile storage resilient to power failure such as a fixed physical address location on flash device itself helps to bootstrap recovery. This can be performed periodically or depending on the required consistency model. Moreover, since GTD size is very small (4KB for 1GB flash), the overhead involved in terms of both space as well as extra operations is also very small. However, at the time of power failure there may be

some mappings present in the Cached Mapping Table, that have been updated but not yet written back to map pages on flash. If strong consistency is required then even the Cached Mapping Table needs to be saved along with the GTD.

3.5 Comparison of Existing State-of-the-art FTLs with DFTL

	Replacement Block FTL [3]	BAST [4]	FAST [23]	SuperBlock [16]	LAST [24]	DFTL	Ideal Page FTL
FTL type	Block	Hybrid	Hybrid	Hybrid	Hybrid	Page	Page
Mapping Granularity	Block	DB-Block LB - Page	DB-Block LB-Page	SB - Block LB/Blocks within SB-Page	DB/Sequential LB - Block Random LB - Page	Page	Page
Division of Update Blocks (M)	-	-	1 Sequential + (M-1) Random	-	(m) Sequential- (M-m) Random (Hot and Cold)	-	-
Associativity of Blocks (Data:Update)	(1:K)	(1:M)	Random LB-(N:M-1) Sequential LB-1:1	(S:M)	Random LB-(N:M-m) Sequential LB-(1:1)	(N:N)	(N:N)
Blocks available for updates	Replacement Blocks	Log Blocks	Log Blocks	Log Blocks	Log Blocks	All Data Blocks	All Blocks
Full Merge Operations	Yes	Yes	Yes	Yes	Yes	No	No

Table 2: FTL Schemes Classification. N: Number of Data Blocks, M: Number of Log Blocks, S: Number of Blocks in a Super Block, K: Number of Replacement Blocks. DB: Data Block, LB: Log Block, SB: Super Block. In FAST and LAST FTLs, random log blocks can be associated with multiple data blocks.

Table 2 shows some of the salient features of different FTL schemes. The DFTL architecture provides some intrinsic advantages over existing state-of-the-art FTLs which are as follows:

- **Full Merge -** Existing hybrid FTL schemes try to reduce the number of full merge operations to improve their performance. DFTL, on the other hand, completely does away with full merges. This is made possible by page-level mappings which enable relocation of any logical page to any physical page on flash while other hybrid FTLs have to merge page-mapped log blocks with block-mapped data blocks.
- **Partial Merge -** DFTL utilizes page-level temporal locality to store pages which are accessed together within same physical blocks. This implicitly separates hot and cold blocks as compared to LAST and Superblock schemes [16, 24] require special external mechanisms to achieve the segregation. Thus, DFTL adapts more efficiently to changing workload environment as compared with existing hybrid FTL schemes.
- **Random Write Performance -** As is clearly evident, it is not necessarily the random writes which cause poor flash device performance but the intrinsic shortcomings in the design of hybrid FTLs which cause costly merges (full and partial) on log blocks during garbage collection. Since DFTL does not require these expensive full-merges, it it is able to improve random write performance of flash devices.

- **Block Utilization -** In hybrid FTLs, only log blocks are available for servicing update requests. This can lead to low block utilization for workloads whose working-set size is smaller than the flash size. Many data blocks will remain un-utilized (hybrid FTLs have block-based mappings for data blocks) and unnecessary garbage collection will be performed. DFTL solves this problem since updates can be performed on any of the data blocks.

4 The FlashSim Simulator

In order to study the performance implications of various FTL schemes, we develop a simulation framework for flash based storage systems called FlashSim. FlashSim is built by enhancing Disksim [7], a well-regarded disk drive simulator. Disksim is an event-driven simulator which has been extensively used in different studies [9, 21] and validated with several disk models. It simulates storage-system components including disk drives, controllers, caches, and various interconnects etc. However, it does not allow the modeling of flash based devices.

FlashSim is designed with a modular architecture with the capability to model a holistic flash-based storage environment. It is able to simulate different storage sub-system components including device drivers, controllers, caches, flash devices, and various interconnects. In our integrated simulator, we add the basic infrastructure required for implementing the internal operations (page read, page write, block erase etc.) of a flash-based device. The core FTL engine is implemented to provide virtual-to-physical address translations along with a garbage collection mechanism. Furthermore, we implement a multitude of FTL schemes: (i) a block-based FTL scheme (replacement-block FTL [3]), (ii) a state-of-the-art hybrid FTL (FAST [23]) , (iii) our page-based DFTL scheme and (iv) an idealized page-based FTL. This setup is used to study the impact of various FTLs on flash device performance and more importantly on the components in the upper storage hierarchy. FlashSim has been validated against a 32GB 2.5" SATA Solid State Drive from Super-Talent [2]. Currently, the simulator has been designed and validated for SLC-based flash devices. The experimental setup is designed with a single plane SLC-based flash memory. However, as part of future study, the existing architecture would be extended to simulate MLC-based flash devices. However, owing to space constraints, we do not present our validation methodology. [1]

[1]Reviewers interested in FlashSim's source code may approach us through the conference chairs. We plan to make it freely available when the constraints of double-blind reviewing do not apply.

5 Experimental Results

We use FlashSim to evaluate the performance of DFTL and compare it with both (i) a state-of-the-art hybrid FTL FAST [23]) and an (ii) idealized page-based FTL with sufficient SRAM (called *Baseline* FTL henceforth).

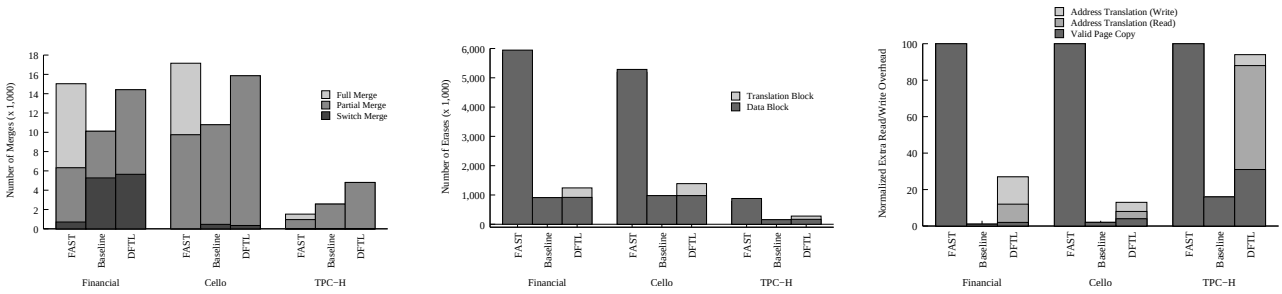
5.1 Evaluation Setup

We simulate a 32GB NAND flash memory with specifications shown in Table 1. To conduct a fair comparison of different FTL schemes, we consider only a portion of flash as the *active region* which stores our test workloads. The remaining flash is assumed to contain cold data or free blocks which are not under consideration. We assume the SRAM to be just sufficient to hold the address translations for FAST FTL. Since the actual SRAM size is not disclosed by device manufacturers, our estimate represents the minimum SRAM required for the functioning of a typical hybrid FTL. We allocate extra space (approximately 3% of the total active region [16]) for use as log-buffers by the hybrid FTL. We exploit intra-request I/O parallelism by adopting a *striping technique* [17] (striping level 4) that splits the requests across multiple channels to be handled simultaneously.

Workloads	Average Request Size (KB)	Read (%)	Sequentiality (%)	Average Request Inter-arrival Time (ms)
Financial (OLTP) [31]	4.38	9.0	2.0	133.50
Cello99 [11]	5.03	35.0	1.0	41.01
TPC-H (OLAP) [42]	12.82	95.0	18.0	155.56
Web Search [40]	14.86	99.0	14.0	9.97

Table 3: Enterprise-Scale Workload Characteristics.

Workloads. We use a mixture of real-world and synthetic traces to study the impact of different FTLs on a wide spectrum of enterprise-scale workloads. Table 3 presents salient features of our workloads. We employ a write-dominant I/O trace from an OLTP application running at a financial institution [31] made available by the Storage Performance Council (SPC), henceforth referred to as the *Financial trace*. We also experiment using Cello99 [11], which is a disk access trace collected from a time-sharing server exhibiting significant writes; this server was running the HP-UX operating system at Hewlett-Packard Laboratories. We consider two read-dominant workloads to help us assess the performance degradation, if any, suffered by DFTL in comparison with other state-of-the-art FTL schemes due to its address translation overhead. For this purpose, we use TPC-H [42], which is an ad-hoc, decision-support benchmark (OLAP workload) examining large volumes of data to execute complex database queries. Also, we use a read-dominant Web Search engine trace [40] made available by SPC.



(a) Merge Operations (b) Block Erases (c) Extra Read/Write Operations

Figure 12: Overheads with different FTL schemes. We compare DFTL with FAST and Baseline for three workloads: Financial, Cello99, and TPC-H. The overheads for the highly read-oriented Web Search workload are significantly smaller than others and we do not show them here. In (c), Address Translation (Read) and Address Translation (Write) denote the extra read and write operations for address translations required in DFTL, respectively. All extra read/write operations have been normalized with respect to FAST FTL scheme.

Finally, we also use a number of synthetic traces to study the behavior of different FTL schemes for a wider range of workload characteristics than those exhibited by the above real-world traces.

Performance metrics. The *device service time* is a good metric for estimating FTL performance since it captures the overheads due to both garbage collection and address translation. However, it does not include the *queuing delays* for requests pending in I/O driver queues. In this study, we utilize both (i) indicators of the garbage collector's efficacy and (ii) response time as seen at the I/O driver (this is the sum of the device service time and time spent waiting in the driver's queue, we will call it the *system response time*) to characterize the behavior/performance of the FTLs.

5.2 Analysis of Garbage Collection and Address Translation Overheads

The garbage collector may have to perform merge operations of various kinds (switch, partial, and full) while servicing update requests. Recall that merge operations pose overheads in the form of block erases. Additionally, merge operations might induce copying of valid pages from victim blocks—a second kind of overhead. We report both these overheads as well as the different kinds of merge operations in Figure 12 for our workloads. As expected from Section 3 and corroborated by the experiments shown in Figure 12, read-dominant workloads (TPC-H and Web Search)—with their small percentage of write requests—exhibit much smaller garbage collection overheads than Cello99 or Financial trace. The number of merge operations and block erases are so small for the highly read-dominant Web Search trace that we do not show these in Figures 12(a),(b), and (c).

Switch merges. Hybrid FTLs can perform switch merges only when the victim update block (selected by garbage collector) contains valid data belonging to logically consecutive pages. DFTL, on the other hand, with its page-based address translation, does not have any such restriction. Hence, *DFTL shows a higher number of switch merges* for even random-write dominant Financial trace as seen in Figure 12(a).

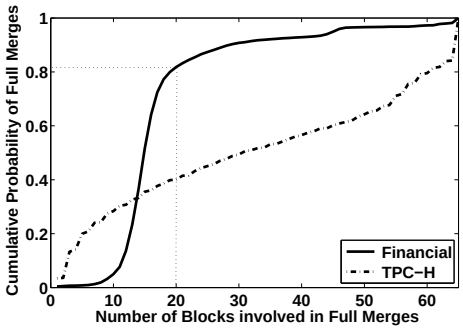
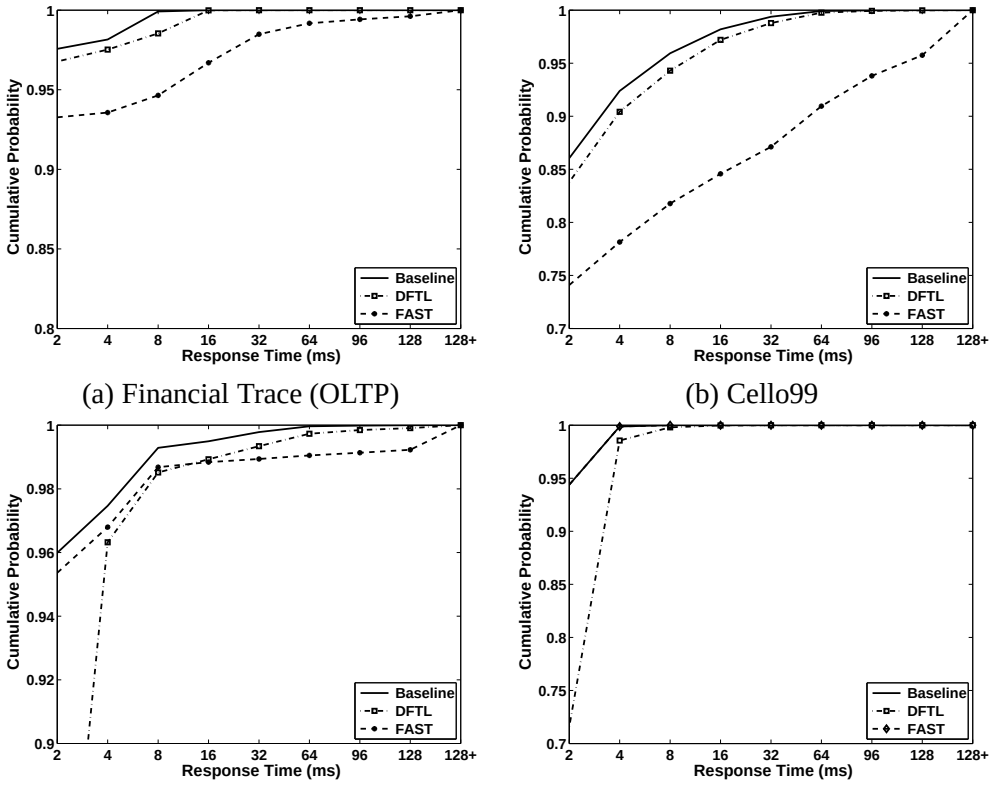


Figure 13: Expensive full merge in FAST FTL. About 20% of full merges involve 20 data blocks or more for the Financial trace.

Full merges. As shown in Figure 13, with FAST, about 20% of the full merges in the Financial trace involve 20 data blocks or more. This is because state-of-the-art hybrid FTLs allow high associativity of log blocks with data blocks while maintaining block-based mappings for data blocks, thus requiring a costly operation of merging data pages in the victim log block with their corresponding data blocks (recall Figure 7 in Section 2). For TPC-H, although DFTL shows a higher number of total merges, its fine-grained addressing enables it to *replace full merges with less expensive partial merges.* With FAST as many as 60% of the full merges involve more than 20 data blocks. As we will observe later, this directly impacts FAST's overall performance.

Figure 12(b) shows the higher number of block erases with FAST as compared with DFTL for all our workloads. This can be directly attributed to the large number of data blocks that need to be erased to complete the full merge operation in hybrid FTLs. Moreover, in hybrid FTLs only a small fraction of blocks (log blocks) are available as update blocks, whereas DFTL allows all blocks to be used for servicing update requests. This not only improves the block utilization in our scheme as compared with FAST but also contributes in reducing the invocation of the garbage collector.

Translation and valid page copying overheads. DFTL introduces some extra overheads due to its address translation mechanism (due to missed mappings that need to be brought into the SRAM from flash). Figure 12(c) shows the normalized overhead (with respect to FAST FTL) from these extra read and write operations along



(a) Financial Trace (OLTP) (b) Cello99

(c) TPC-H (d) Web-Search

Figure 14: Each graph shows the Cumulative Distribution Function (CDF) of the average system response time for different FTL schemes.

with the extra valid pages required to be copied during garbage collection. Even though the address translation accounts for approximately 90% of the extra overhead in DFTL for most workloads, overall it still performs less extra operations than FAST. For example, DFTL yields a 3-fold reduction in extra read/write operations over FAST for the Financial trace. Our evaluation supports the key insight behind DFTL, namely that the temporal locality present in workloads helps keep this address translation overhead small, i.e., most requests are serviced from the mappings in SRAM. DFTL is able to utilize page-level temporal locality in workloads to reduce the valid page copying overhead since most hot blocks (data blocks and translation blocks) contain invalid pages and are selected as victims by our garbage collector. In our experiments, we observe about 63% hits for address translations in SRAM for the financial trace even with our conservatively chosen SRAM size. In a later sub-section, we investigate how this overhead reduces further upon increasing the SRAM size.

Workloads	FTL Type	System Response Time		Device Response Time		I/O driver Queuing Delay	
		Average (ms)	std.dev	Average (ms)	std.dev	Average (ms)	std.dev
Financial	Baseline	0.43	0.81	0.39	0.79	0.04	0.19
	FAST	2.75	19.77	1.67	13.51	1.09	13.55
	DFTL	0.61	1.52	0.55	1.50	0.06	0.29
Cello99	Baseline	1.50	4.96	0.41	0.80	1.08	4.88
	FAST	16.93	52.14	2.00	14.59	14.94	50.20
	DFTL	2.14	6.96	0.59	1.04	1.54	6.88
TPC-H	Baseline	0.79	2.96	0.68	1.78	0.11	2.13
	FAST	3.19	29.56	1.06	11.65	2.13	26.74
	DFTL	1.39	7.65	0.95	2.88	0.44	6.57
Web Search	Baseline	0.86	0.64	0.68	0.44	0.18	0.46
	FAST	0.86	0.64	0.68	0.44	0.18	0.46
	DFTL	1.24	1.06	0.94	0.68	0.30	0.78

Table 4: Performance metrics for different FTL schemes with enterprise-scale workloads.

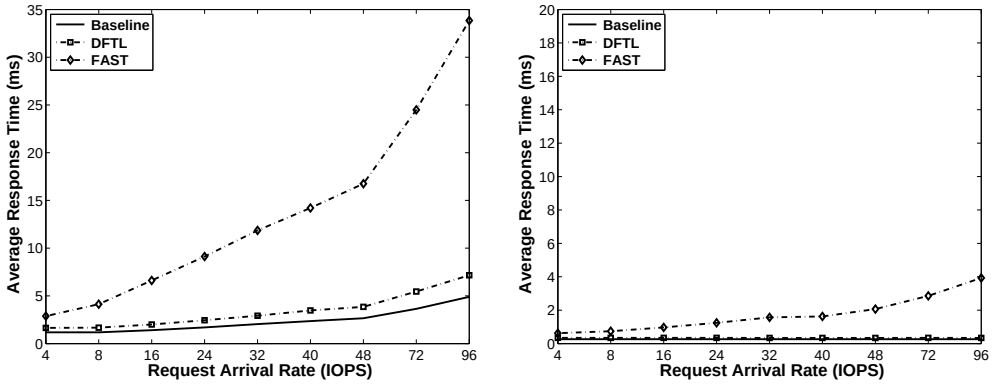
5.3 Performance Analysis

Having seen the comparison of the overheads of garbage collection and address translation for different FTLs, we are now in a position to appreciate their impact on the performance offered by the flash device. The Cumulative Distribution Function of the average system response time for different workloads is shown in Figure 14. DFTL is able to closely match the performance of Baseline scheme for the Financial and Cello99 traces. In case of the Financial trace, DFTL reduces the total number of block erases as well as the extra page read/write operations by about 3 times. This results in improved device service times and shorter queuing delays (refer to Table 4) which in turn improve the overall I/O system response time by about 78% as compared to FAST.

For Cello99, the improvement is much more dramatic because of the high I/O intensity which increases the pending requests in the I/O driver queue, resulting in higher latencies. Reviewers should be careful about the following while interpreting these results: we would like to point out that Cello99 represents only a point within a much larger enterprise-scale workload spectrum for which the gains offered by DFTL are significantly large. More generally, DFTL is found to improve the average response times of workloads with random writes with the degree of improvement varying with the workload's properties.

For read-oriented workloads, DFTL incurs a larger additional address translation overhead and its performance deviates from the Baseline (Figure 14(c) & (d)). Since FAST is able to avoid any merge operations in the Web search trace, it provides performance comparable to Baseline. However, for TPC-H, it exhibits a *long tail* primarily because of the expensive full merges and the consequent high latencies seen by requests in the I/O driver queue.

Hence, even though FAST services about 95% of the requests faster, it suffers from long latencies in the remaining requests, resulting in a higher average system response time than DFTL.



(a) Workload A (b) Workload B

Figure 15: Performance comparison of various FTLs with changing I/O intensity for synthetic workloads. DFTL is able provide improved performance as well as sustain overloaded behavior in workloads much better than FAST. The 99% confidence intervals are very small and hence not shown.

5.4 Exploring a Wider Range of Workload Characteristics

We have seen the improvement in performance for different realistic workloads with DFTL as compared to state-of-the-art FTLs. Here, we widen the spectrum of our investigation by varying one workload property, namely I/O request arrival intensity. An enterprise-scale FTL scheme should be robust enough to sustain periods of increased I/O intensity, especially for write dominant workloads. In order to simulate such changing environments we use two synthetic workloads with varying characteristics: Workload A is predominantly random write-dominant whereas Workload B has a large number of sequential writes. With increasing request arrival rate, the flash device transitions from a *normal operational region* to an *overloaded region*.

As shown in Figure 15, for Workload A the transition into overloaded region is marked by very high gradient in response times pointing to the un-sustainability of such an environment using FAST. On the other hand, DFTL is not only able to provide improved performance in the operational region but is also able to sustain higher intensity of request arrivals. It provides *graceful degradation* in performance to sustained increase in I/O intensity, a behavior especially desirable in enterprise-scale systems. For sequential workload B, the merge overhead is reduced because of higher number of switch merges as compared to full-merges. Thus, FAST is able to endure the increase in request arrival rate, much better than its own performance with random-write dominant workload A. However, we still observe better performance from DFTL, which is able to approximate the performance of

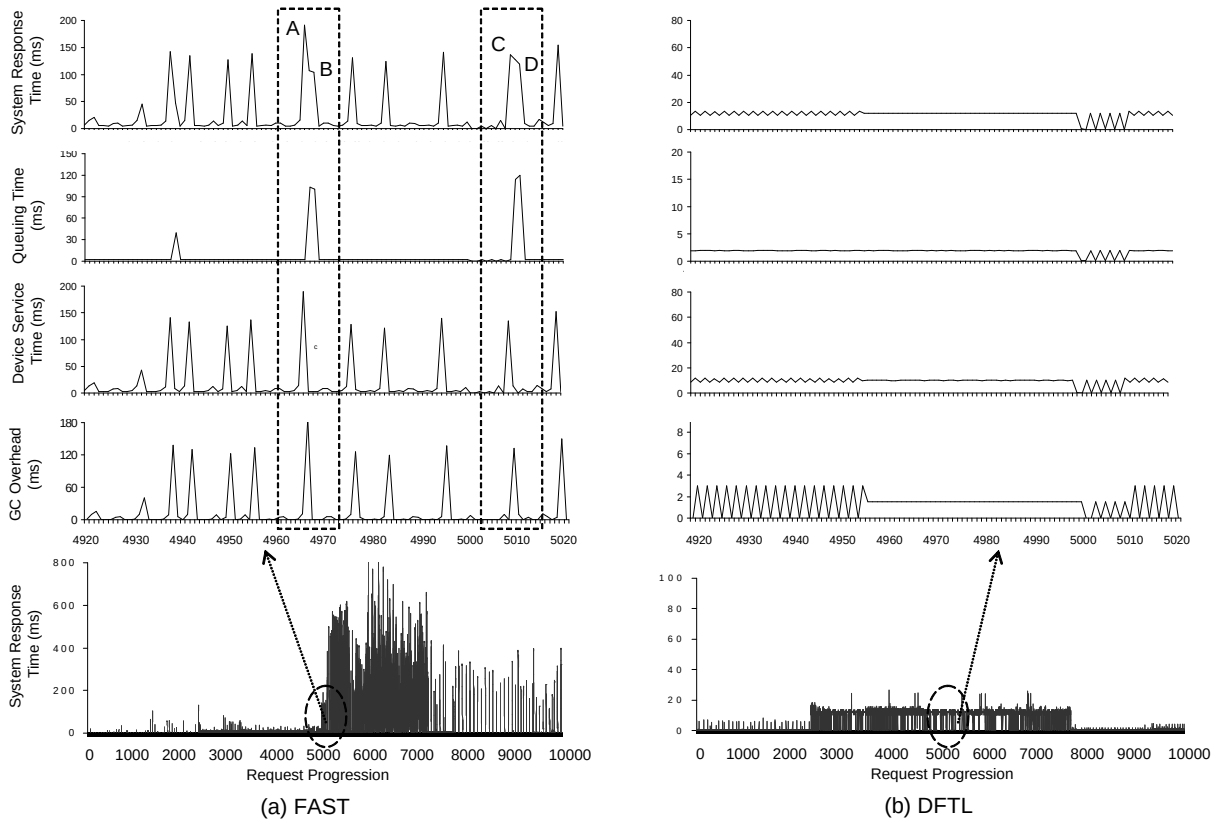


Figure 16: Microscopic analysis of DFTL and FAST FTL schemes with Financial trace. The selected region (requests 4920 to 5020) represents transition from normal operational region to overloaded region. Requests A & C undergo full-merges in FAST. However, their impact is also seen on requests B & D through long queuing latencies. Meanwhile, DFTL is able to provide much better performance in the same region.

Baseline scheme because of the availability of all blocks to service the update requests.

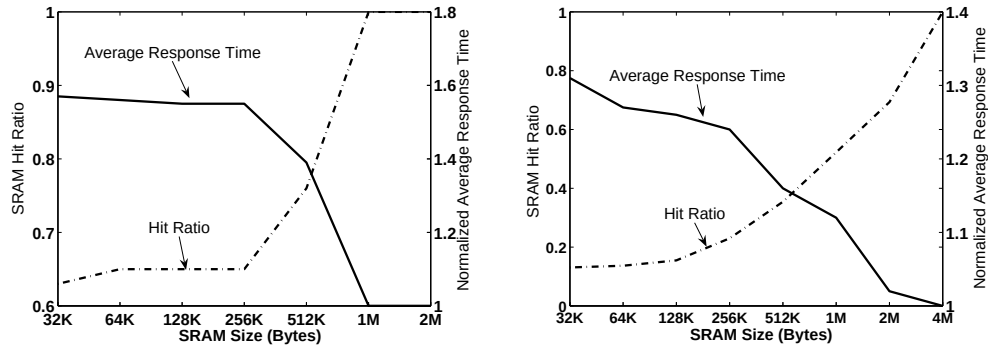
5.5 Microscopic Analysis

In this sub-section, we try to perform a microscopic analysis of the impact of garbage collection on instantaneous response times by installing probes within FlashSim to trace individual requests.

Figure 16 represents a same set of 100 consecutive requests being serviced by FAST and DFTL for the Financial trace. This region illustrates transition from a sustainable I/O intensity (operational region) to a period of very intense I/Os (overloaded region) in the Financial trace. As is clearly visible, FAST suffers from higher garbage collection overhead and requests undergo higher latencies as compared to DFTL. Full merges cause a large number valid pages to be copied and the corresponding blocks to be erased. This results in higher device service time for the request undergoing these operations. This in turn causes the pending requests in the I/O driver queue to incur longer latencies. Thus, even though the device service time for these requests is small; the overall

system response time increases. For example, in the top highlighted region in Figure 16, request A undergoes full merge resulting in very high device service time. While A is being serviced, the pending request B incurs high latency in the I/O driver queue (spike in queueing time for B) which increases its overall system response time. The same phenomenon is visible for requests C and D. Thus, full merges not only impact the current requests but also increase the overall service times for subsequent requests by increasing queuing delays. In sharp contrast, during the same period, DFTL is able to keep garbage collection overhead low and provide sustained improved performance to the requests as it does not incur any such costly full merge operations.

5.6 Impact of SRAM size



(a) Financial Trace (b) TPC-H Benchmark

Figure 17: Impact of SRAM size on DFTL. Response times have been normalized with respect to the Baseline FTL scheme. For both the Financial trace and TPC-H, there is performance improvement with increased SRAM hit-ratio. However, beyond the working-set size of workloads there is no benefit of additional SRAM for address translation. The 99% confidence intervals are very small and hence not shown.

All the experiments in the preceding subsections were done by utilizing the bare minimum amount of SRAM necessary for implementing any state-of-the-art hybrid FTL scheme. Even with this constrained SRAM size, we have shown that DFTL outperforms the existing FTL schemes for most workloads. The presence of temporal locality in real workloads reduces the address-translation overhead considerably. Figure 17 shows the impact of increased available SRAM size on DFTL. As seen, greater SRAM size improves the hit ratio, reducing the address translation overhead in DFTL, and thus improving flash device performance. As expected, with the SRAM size approaching the working set size (SRAM hit ratio reaches 100%), DFTL's performance becomes comparable to Baseline. Increasing SRAM size for holding address translations beyond the workload working-set size does not provide any tangible performance benefits. It would be more beneficial to utilize this extra SRAM for caching popular read requests, buffering writes, etc. than for storing unused address translations.

6 Concluding Remarks and Future Directions

We argued that existing FTL schemes, all based on storing a mix of page-level and block-level mappings, exhibit poor performance for enterprise-scale workloads with significant random write patterns. We proposed a complete paradigm shift in the design of the FTL with our Demand-based Flash Translation Layer (DFTL) that selectively caches page-level address mappings. Our experimental evaluation using a comprehensive flash simulator called FlashSim with realistic enterprise-scale workloads endorsed DFTL's efficacy for enterprise systems by demonstrating that DFTL offered (i) improved performance, (ii) reduced garbage collection overhead, (iii) improved overload behavior and (iv) most importantly unlike existing hybrid FTLs is free from any tunable parameters. As a representative example, a predominantly random write-dominant I/O trace from an OLTP application running at a large financial institution showed a 78% improvement in average response time due to a 3-fold reduction in garbage collection induced operations as compared to a state-of-the-art FTL scheme. For the well-known read-dominant TPC-H benchmark, despite introducing additional operations due to mapping misses in SRAM, DFTL improved response time by 56%.

As part of further validation, we plan to evaluate the efficacy of DFTL in consolidated enterprise-scale environments using mixes of disparate workloads. Another direction of ongoing research studies the feasibility of hybrid storage systems employing flash at appropriate places within the enterprise storage hierarchy along with hard disk drives.

References

[1] Samsung 256GB Flash SSD With High-Speed Interface . `http://www.i4u.com/article17560.html`.

[2] 2.5" Super-Talent SATA Solid State Drive. `http://www.supertalent.com/products/ssd-commercial.php?type=SATA`.

[3] Amir Ban. Flash File System. In *United States Patent, No 5,404,485*, 1993.

[4] T. Chung, D. Park, S. Park, D. Lee, S. Lee, and H. Song. System Software for Flash Memory: A Survey. In *Proceedings of the International Conference on Embedded and Ubiquitous Computing*, pages 394–404, August 2006.

[5] E. Gal and S. Toledo. Algorithms and Data Structures for Flash Memories. *ACM Computing Survey*, 37(2):138–163, 2005.

[6] Flash Drives Hit by High Failure Rates. `http://www.techworld.com/storage/news/index.cfm?newsid=11747`.

[7] G.R. Ganger, B.L. Worthington, and Y.N. Patt. *The DiskSim Simulation Environment Version 3.0 Reference Manual*.

[8] A. Gulati, A. Merchant, and P. J. Varman. pClock: An Arrival Curve based Approach for QoS Guarantees in Shared Storage Systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 13–24, June 2007.

[9] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. DRPM: Dynamic Speed Control for Power Management in Server Class Disks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 169–179, June 2003.

[10] J. Hennessy and D. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 2003.

[11] HP Labs. Tools and Traces. `http://tesla.hpl.hp.com/public_software/`.

[12] Intel, STMicroelectronics Deliver Industry's First Phase Change Memory Prototypes. `http://www.intel.com/pressroom/archive/releases/20080206corp.htm`.

[13] Intel StrataFlash Memory Technology Overview. `http://download.intel.com/technology/itj/q41997/pdf/overview.pdf`.

[14] J. Kim, J.M. Kim, S.H. Noh, S. Min, and Y. Cho. A Space-Efficient Flash Translation Layer for Compactflash Systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.

[15] D. Jung, Y. Chae, H. Jo, J. Kim, and J. Lee. A Group-based Wear-Leveling Algorithm for Large-Capacity Flash Memory Storage Systems. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 160–164, September 2007.

[16] J. Kang, H. Jo, J. Kim, and J. Lee. A Superblock-based Flash Translation Layer for NAND Flash Memory. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 161–170, October 2006.

[17] J. Kang, J. Kim, C. Park, H. Park, and J. Lee. A Multi-Channel Architecture for High-Performance NAND Flash-based Storage System. *Journal of System Architecture*, 53(9):644–658, 2007.

[18] R. Karedla, J. Spencer Love, and Bradley G. Wherry. Caching Strategies to Improve Disk System Performance. *IEEE Transactions on Computer*, 27(3):38–46, 1994.

[19] A. Kawaguchi, S. Nishioka, and H. Motoda. A Flash-Memory based File System. In *Proceedings of the Winter 1995 USENIX Technical Conference*, pages 155–164, 1995.

[20] H. Kim and S. Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, Feburary 2008.

[21] Y. Kim, S. Gurumurthi, and A. Sivasubramaniam. Understanding the Performance-Temperature Interactions in Disk I/O of Server Workloads. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, Febuary 2006.

[22] S. Lee and B. Moon. Design of Flash-based DBMS: An In-Page Logging Approach. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 55–66, August 2007.

[23] S. Lee, D. Park, T. Chung, D. Lee, S. Park, and H. Song. A Log Buffer based Flash Translation Layer Using Fully Associative Sector Translation. *IEEE Transactions on Embedded Computing Systems*, 6(3):18, 2007.

[24] S. Lee, D. Shin, Y. Kim, and J. Kim. LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems. In *Proceedings of the International Workshop on Storage and I/O Virtualization, Performance, Energy, Evaluation and Dependability (SPEED2008)*, Feburary 2008.

[25] A. Leventhal. Flash Storage Memory. *Communications of the ACM*, 51(7):47–51, 2008.

[26] K. M. J. Lofgren, R. D. Norman, G B. Thelin, and A. Gupta. Wear Leveling Techniques for Flash EEPROM. In *United States Patent, No 6,850,443*, 2005.

[27] Micron 16GB Mass Storage. `http://www.micron.com/products/partdetail?part=MT29F16G08DAAWP`.

[28] MTRON. `http://www.mtron.net/`.

[29] H. Niijima. Design of a Solid-State File Using Flash EEPROM. *IBM Journal of Research and Developement*, 39(5):531–545, 1995.

[30] A. Nitin, P. Vijayan, and W. Ted. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference*, June 2008.

[31] OLTP Trace from UMass Trace Repository. `http://traces.cs.umass.edu/index.php/Storage/Storage`.

[32] RamSan-500, Texas Memory Systems. `http://www.ramsan.com/products/ramsan-500/`.

[33] S. Tehrani, J.M. Slaughter, E. Chen, M. Durlam, J. Shi, and M. DeHerren. Progress and Outlook for MRAM Technology. *IEEE Transactions on Magnetics*, 35(5):2814–2819, 1999.

[34] Samsung NAND Flash K9F5608U0M. `http://www.samsung.com/Products/Semiconductor/index.htm`.

[35] Samsung's Q30-SSD Laptop with 32GB flash drive. `http://www.samsung.com/`.

[36] Y. Shimada. FeRAM: Next Generation Challenges and Future Directions. *Electronics Weekly*, May 2008.

[37] Symmetrix DMX-4, EMC. `http://www.emc.com/products/detail/hardware/symmetrix-dmx-4.htm`.

[38] Technical Report (TN-29-07): Small-Block vs. Large-Block NAND Flash Devices. `http://www.micron.com/products/nand/technotes`.

[39] TPC-C, an OLTP Benchmark. `http://www.tpc.org/tpcc/`.

[40] Websearch Trace from UMass Trace Repository. `http://traces.cs.umass.edu/index.php/Storage/Storage`.

[41] White Paper: Datacenter SSDs: Solid Footing for Growth. `http://www.samsung.com/global/business/semiconductor/products/flash/FlashApplicationNote.html`.

[42] J. Zhang, A. Sivasubramaniam, H. Franke, N. Gautam, Y. Zhang, and S. Nagar. Synthesizing Representative I/O Workloads for TPC-H. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2004.